University of Antwerp

Bachelor in Computer Science

Bachelor Dissertation

cDEVS Project Evaluation Report Final Version

Authors:
Nathan Steurs
Ian Vermeulen
David Dansaert
Joren Van de Vondel
Lotte Vergauwen

Supervisor:
Kurt Van Mechelen
Jan Broeckhove

June 15, 2015

cDevs

Contents

1 Introduction

cDEVS is a C++11 port of the pypdevs[1] project, originally written in Python. The main goal of this project is to provide a performant, DEVS-formalism compliant, Discrete Event Simulator, based on the pypdevs project, whilst making use of all the benefits C++(11) has to offer. In this document you will find an overview of all features implemented, alongside the design decisions that make them come to fruition. We would like to thank Kurt van Mechelen and Jan Broeckhove for their assistance where needed throughout the development of the cDevs.

2 Previous Versions

2.1 Alpha Functionality

- No outwardly visible functionality available
- Solver not implemented
- Basic implementation of components (tracer, scheduler, ...) available
- General code structure available (cfr. UML)

2.2 Beta Functionality

2.2.1 Completed Features

- Classic DEVS has been implemented
- Basic tracer (Verbose) and XML-tracer implemented
- GVT calculation implemented (Needs testing!)
- Added warnings, Valgrind, Cppcheck and test result information to Jenkins CI-server
- Modelloader functional
- TrafficLight_Classic ported from pypDEVS

2.2.2 Features to come

- User Manual (Will be attached to this document)
- Diverse Schedulers and Tracers (JSON)
- Complete C++11 compliant implementation of <memory>
- **Parallelism**
- (More) Unit Testing & Scenario Testing
- Checkpointing and Rollback
- Thorough comparison of Time/Space performance with Pypdevs and ADEVS

[1]More information can be found here:`https://bitbucket.org/comp/pypdevs` and here: `http://msdl.cs.mcgill.ca/projects/DEVS/PythonPDEVS/`

3 Final Version - June 15th 2015

The following list contains a concise view of implemented features for the final version of the cDEVS project:

3.1 Features

- Classic Devs Simulation
- Parallel Simulation using Optimistic Time (Mattern's Algorithm)
- Tracing in JSON/XML/Verbose format, with customization possibilities.
- Heap & List Schedulers
- C++11 Smart Pointer implementation
- Broad testing
- Benchmarking
- Performance comparison between cDEVS and pypdevs/ADEVS.

4 Issues during development

4.1 Team Communication - Management of a Large Project

For all of us, cDevs was the first "real" project we had ever started working on. We have had quite a few bumps along the way to get to where we are now. The first major hurdle to overcome was definitely creating a good work flow for the team, through proper communication. Improper communication (or a lack thereof) soon led to issues during the first phase of the project, when creating the important design documents. These problems were reported to Kurt van Mechelen, and were discussed during meetings, and resolved for the time being. During the implementation phase of the project, issues similar to those of the first phase, were quick to re-emerge. Lack of communication caused a backlog of 2 weeks, which lead to a complete default of the Alpha Presentation deadline, as can be read above. We, again, sat down with Kurt van Mechelen, and devised a better way of working. From this point onwards, a complete overview of each persons work flow was maintained using a spreadsheet, documenting all deadlines, milestones, etc... (something that should have been done at an earlier time in the project). During the Beta presentation we delivered what should have been delivered at Alpha, this time, with a clear plan of how to combat the future of the project. Since then, we have implemented all features we had planned for during the Beta Evaluation. From this, we have learned that working as a team, is not as easy as it seems. For future projects, it is vital that a proper work software engineering methodology (such as Scrum) is adapted to the project, to avoid problems like these.

4.2 Initial Structure Port

In the initial stages of the development, we faced quite a few problems while transitioning from Python to C++. Because of the dynamic typing present in python, there were quite a few pitfalls to overcome when porting. In some cases, methods have different types of return values when called on a certain input. This gave us some overhead in having to make decisions on which data types to use in C++, or in having to split up certain functionality because it was unable to provide the same interface in C++.

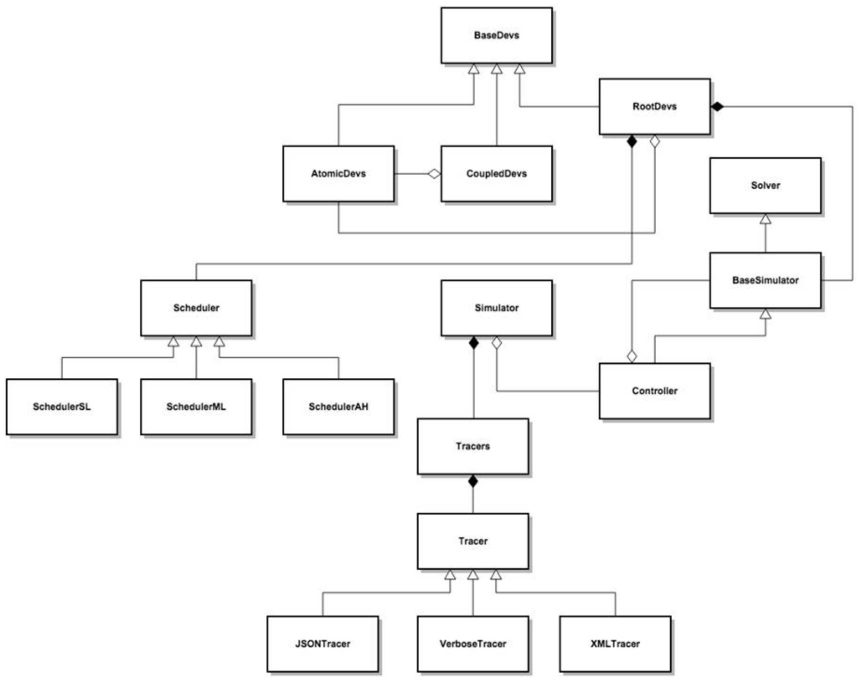
4.3 Transitioning from raw to smart pointers

Up until the Beta presentation, the cDEVS project was written without making use of C++11 ¡memory¿. The initial transition from raw to smart pointers took more time than we originally planned for. Many implementation issues arose, which were not accounted for during the prototyping, specifically with the use of std::enable_shared_from_this::shared_from_this in the constructor. This lead to a very big rewrite of the code base, with implementation of the factory design pattern to avoid using std::shared_from_this in the constructor of a class. However, in the end we are happy to have made the transition as it provides us with proper garbage collection as well as proving a more friendly debugging experience.

5 Design Decisions in cDEVS

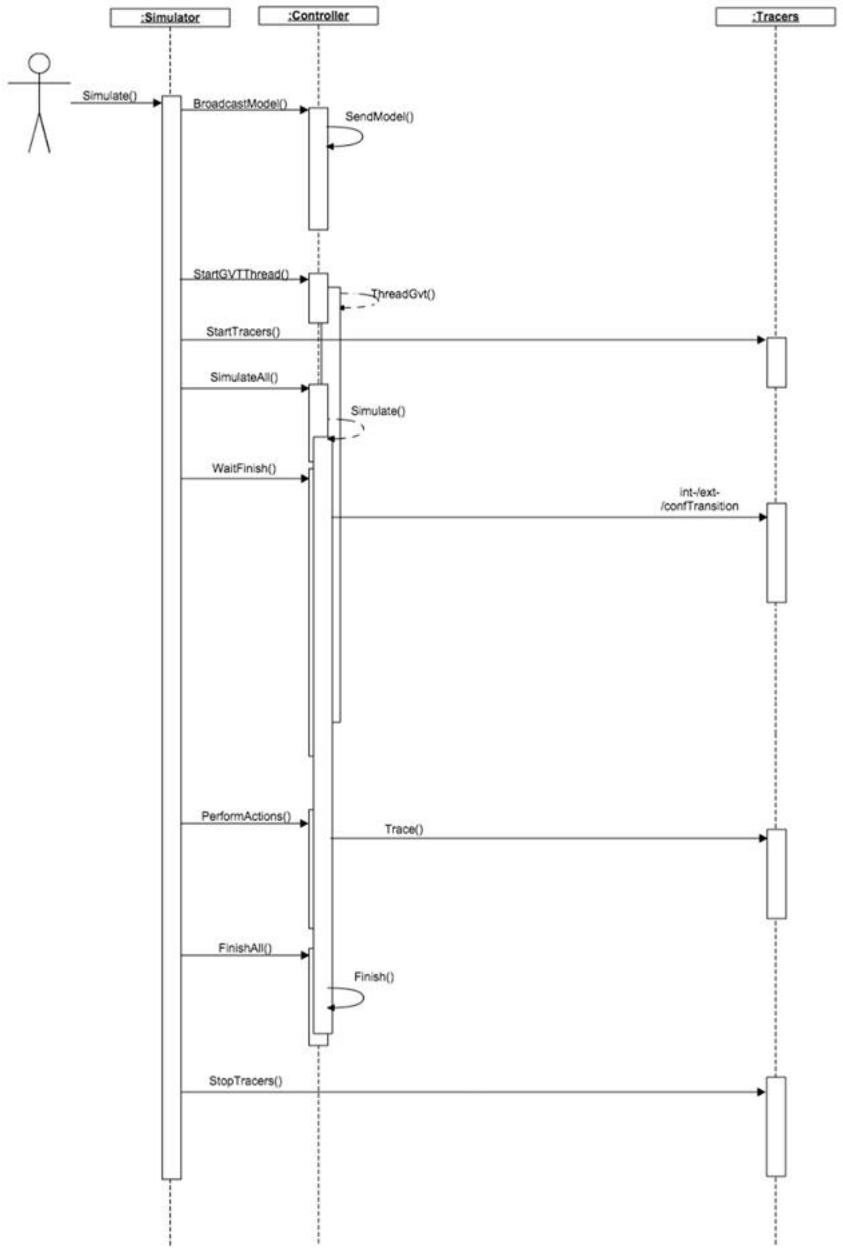
5.1 UML Overview

5.1.1 General cDEVS Structure



This UML schema represents the basic structure of the cDEVS architecture. Solid diamonds represent composition, normal diamonds represent aggregation. We have adhered to the pypdevs design as far as classes and methods are concerned. Apart from a few modifications for some small issues, the only large modification to the existing pypdevs structure is the implementation of the CRTP, about which you can read more below.

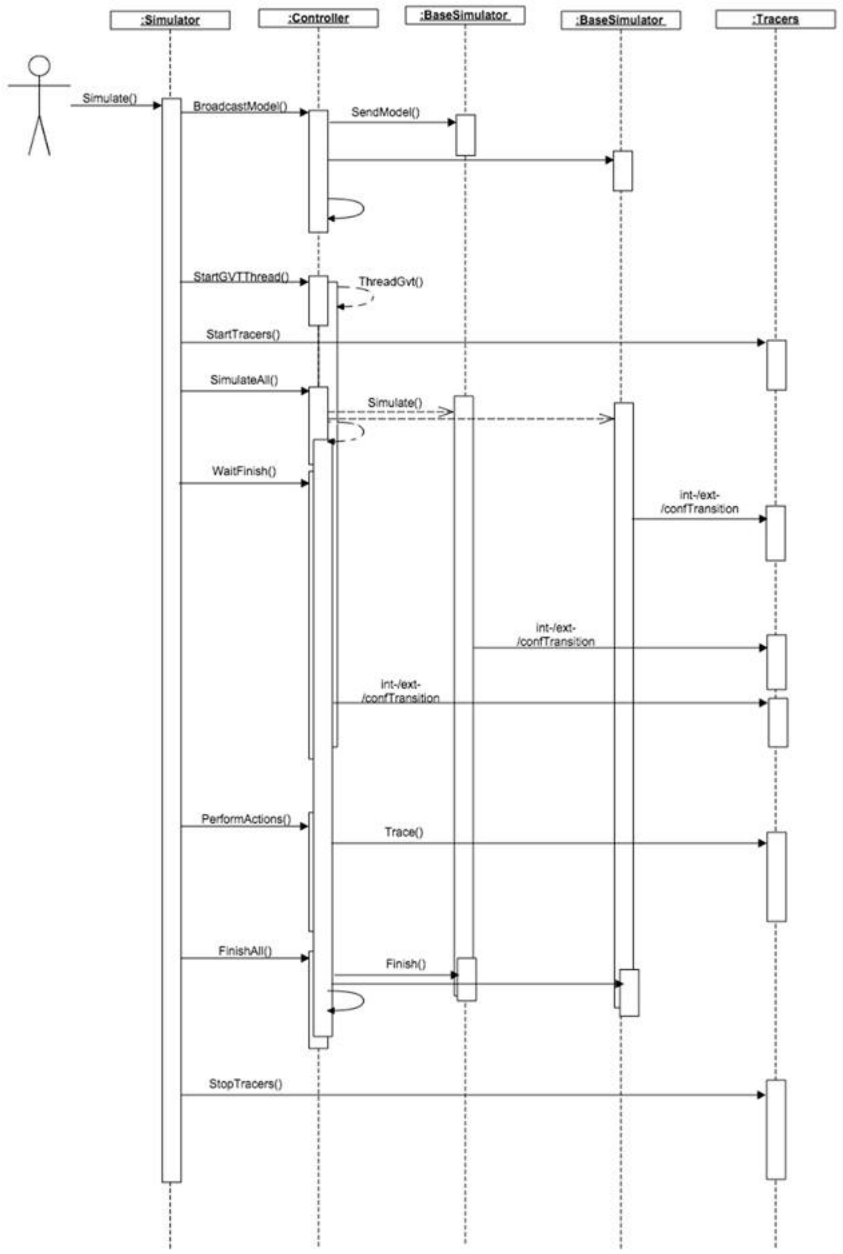
5.1.2 Classic Scenario



The user loads his model and then configures the simulation object. He can set the termination condition, the termination time, the amount of cores, the tracer output, etc... . These are omitted from the UML lifecycle diagram, as they are not part of the simulation itself.

The first step towards simulation is calling the Simulate() method of the Simulator. The Simulator will then call the Controller to prepare to model for simulation (i.e. convert it to a RootDEVS model). Afterwards, the Simulator requests the controller to start the GVT thread. The controller will then call a threaded function ThreadGvt() which will take care of all GVT calculations and eventually, of the checkpointing and fossil collection. After this is done, the tracers will be started by the simulator. Then a new thread for the BaseSimulator::Simulate function is created at the controller, which will take care of the actual simulation. This method will in turn call the tracer to trace the transitions it performs. After the controller is done waiting for the simulation to finish, the GVT thread is shut down, and the actions are performed. This will cause the trace method to be called and the actual output to be printed. Finally, the controller cleans up the simulation by calling FinishAll() which will stop the running simulation thread and stops the tracers.

5.1.3 Parallel Scenario



The simulation starts the same way as classic DEVS does. The Simulator will now call the Controller to broadcast the user model and prepare it for simulation on each of the different kernels. Afterwards, the Simulator requests the controller to start the GVT thread. The controller will then call a threaded function ThreadGvt() which will take care of all GVT calculations and eventually, of the check pointing and fossil collection. After this is done, the tracers will be started by the simulator. Next, the Controller will asynchronously request the different kernels to perform a simulation on their own RootDevs model. All these simulation calls will concurrently call the tracers to perform the tracing for their transitions. After all simulations have finished, the wait finish will return and join the VT thread. After this, the actions are performed, this will cause the trace method to be called and the actual output to be printed. The controller then cleans up the simulation by calling FinishAll() which will command each kernel to finish their running (and waiting) simulation thread. Finally, the Simulator will clean up the tracers by calling StopTracers().

5.2 Design Patterns

5.2.1 Abstract Factory Pattern

The AFP is used in the project to encourage correct use of smart pointers, and to ensure safe use of $std :: shared\_from\_this$, otherwise, one must assume that the user correctly creates the smart pointers, which is not a safe assumption to make. The user is in this way forced to call the create method, which will return a smart pointer to the model for proper use in the simulation.

Internally, we also employ the Factory Pattern[2] in order to ensure correct use of $std :: shared\_from\_this$, to be more precise:

$std :: shared\_from\_this$ requires the existence of a smart pointer to the object. Furthermore, $std :: shared\_from\_this$ can never be used in the constructor, because at this point, a shared pointer to the object does not exist. We employ factories to create objects and their respective smart pointers, then the factory calls the methods that require $std :: shared\_from\_this$, and finally the shared pointer is returned to the user. We chose to encapsulate this process in the factories because we don't want to reveal the internal structure of the project to the user (so they are unable to manipulate it, if they want to).

An alternative solution would have been to invoke these methods at the start of the simulation, before passing anything to the simulator. However, most of these methods encompass structural changes in the model (i.e, connection of ports, hierarchy), which means we would be passing an incomplete model to the simulator, to have the simulator complete the model before simulating it. We decided that it was a cleaner and more logical solution to pass the completed model to the simulator before starting the simulation.

5.2.2 Curiously Recurring Template Pattern

We decided to employ CRTP in the development of cDEVS. CRTP is normally used to enable static polymorphism, literally slashing the cost of looking up the virtual method calls in the vtable, as there is no need for one. However, we do not specifically make use of this advantage of CRTP in the cDEVS project. The main reason why we employ this pattern is to provide a compulsory interface to the user when defining their own classes, and for encapsulation so that internal logic stays internal, and is not visible to the user. When the user defines their own class (for example, a Derived from Base), they must pass Derived as a template argument to the Base (when inheriting). Base will then implement its own methods, using the Derived class as a template parameter.

5.3 Data Structures

For standard library containers, the main decision to be made was whether to use std::vector or std::list. Generally, we employ lists when we have to insert and/or remove from the container a lot (at random locations), and std::vector in other cases. In other cases, std::tuple is preferred over std::pair, for serialization purposes.

5.4 Smart Pointers

Originally, cDevs was implemented using raw pointers only, with no use of the C++11 implementation of smart pointers. This lead to a lot of issues regarding memory, even with careful implementation of the destructors and our own implementation of garbage collection. We made the decision to implement smart pointers in order to maintain proper memory management. The only place where raw pointers are still being used is when loading a model with the model loader, as system calls do not support smart pointers. Special distinction has been made between std::unique_ptr and std::shared_ptr. Whenever an object has only 1 owner, and is solely used and referred to by that owner. When no guarantee can be made about unique ownership, or when the lifetime of an object that would own the std::unique_ptr, is shorter than that of the std::unique_ptr itself, we use std::shared_ptr. Because of the hierarchic structuring, we use shared pointers to maintain the children in the parent, but each child holds an std::weak_ptr in order to avoid cyclic references. Lastly, Ports will hold an std::weak_ptr to another port, in order to avoid double freeing errors.

[2]Here we employ the Factory Pattern, not the AFP.

6 Functional Requirements

6.1 Classic Devs

It is currently possible to simulate a model following the Classic Devs specification. An example of a classic traffic light model can be found in the examples folder. For more information, please refer to the user manual.

6.2 Parallel Devs

Parallel Devs is fully functional, using Optimistic Time Synchronisation with Mattern's Algorithm. An example of a parallel traffic light model can be found in the examples folder. For more information, please refer to the user manual.

6.3 Model loader

The model loader currently offers two different ways of interacting with experiments and user defined models. The first method (of which examples can be found in the examples folder in the project itself) consists of linking the experiments to the cDEVS library which is generated by compiling the project. The second method (of which examples can be found in the test scenarios) consists of loading in the models at runtime.

6.4 Direct Connect

The Direct Connect function is implemented in the same way as pypdevs.

6.5 Serialization

As previously mentioned during the prototyping phase, the header-only library Cereal[3] in employed order to serialize the states of the models. In order to understand how cDEVS employs this library, please refer to the user manual.

6.6 Tracing

Tracing is implemented a little differently from pypdevs. Tracers all inherit from an abstract base class Tracer, which implements 4 mandatory methods. A Verbose format is supported, alongside XML and JSON. For more information regarding these tracers, and how to specify your own custom tracer, please refer to the user manual.

6.7 Scheduling

cDevs currently supports a minimal list scheduler, a sorted list scheduler and an activity heap scheduler.

6.8 Benchmarking

In order to compare performance of cDEVS with pypdevs and ADEVS, several benchmarks have been ported from pypdevs. Currently, a comparison for PHOLD, Devstone and Queue has been made.

[3]More information can be found in our serialization prototype, which can be found here:`http://vandevondel.eu/docs/serialization.pdf`

7 Non-Functional Requirements

In order to monitor the non-functional requirements of the system, we use a series of tools that are integrated in the continuous integration server.

7.1 Correctness

In order to ensure that the output of our simulator is correct, we perform comparisons with output produced by pypdevs. Our scenario testing works in the same way: We simulate a certain model in pypdevs and store its output. Then we do the same with our simulator, and initially compare both outputs by hand. If this output is the same as pypdevs, we store it as correct. During testing, we simulate the same scenario again, and compare it to our previously correct output and report on the result.

7.2 Testing

The final version of the project has 47 tests running, consisting of scenario and unit tests.

7.3 Valgrind

Because the use of memory is very important in our implementation of the simulator, we chose to work with Valgrind as a way to track and resolve memory issues. The Valgrind plugin informs the owner of the repository where their memory issues are, so they can be fixed in the next commit.

7.4 Cppcheck

We have also opted to use a static C++ code analysing tool. Cppcheck is able to detect bad practice and/or errors that the compiler is unable to catch, or notify us incase of possibilities of static performance increases, resulting in increased safeness and performance.

7.5 Platform Independence

cDEVS currently compiles and runs on all 3 major platforms (Unix, OS X and Windows) with the following requirements (-std=c++11 is specified in the CMakeLists to provide the use of C++11):

Unix The project can successfully be compiled with GCC (V4.8) on Unix platforms with use of Cmake.

OS X The project can successfully be compiled with CLANG (V6.1.0) on OS X with the use of Cmake.

Windows The project can successfully be compiled using MinGW-x64. It can be compiled using Cmake and provided CMakeLists.txt files.

7.6 Performance

One of the major considerations of the cDEVS project, and ultimately the most important reason for the port from python to C++, is performance. The next section will go over the performance differences in detail.

8 Performance Comparison: pypdevs and cDEVS

The following table holds all benchmarking information, with the parameters as described. The time measured is total CPU time, the actual runtime can be shorter. All scenarios were run on average 50 times, the output in the table represents the average of all runs.

8.1 Traffic - Queue - PHold - Devstone

•	PYPDEVS	cDEVS	Factor of Improvement
TRAFFIC C	0.116	0.006	+19,9
TRAFFIC P	9.854	0.379	+26
PHOLD 1	15.573	0.323	+48,21
PHOLD 2	45.57	2.349	+19,39
QUEUE	40.949	29.472	+1,39
DEVSTONE	13.511	2.378	+5,66
DEVSTONE LARGE	N/A	27.288	N/A

- TRAFFIC C: Classic Devs, Termination Time = 400.
- TRAFFIC P: 4 Kernels, Termination Time = 10000.
- PHOLD 1: 2 Kernels, 4 Nodes, 10 Atomics/Node, 0 Iterations, 0.1 remotes, Termination Time = 100, GVT interval = 5, CP = 0.
- PHOLD 2: 2 Kernels, 10 Nodes, 25 Atomics/Node, GVT = 5, CP = 1, IT = 0, Remote = 0.1.
- QUEUE: 800 models, Termination Time = 300, GVT Interval = 1, ClassicDevs, CP Interval = 0.
- DEVSTONE: 4 Kernels, width = 30, depth = 30, Termination time = 100.
- DEVSTONE LARGE: 2 Kernels, width = 300, depth = 30, Termination Time = 100, GVT = 5, CP = 1.

From the above results we can clearly see that the C++ implementation is greatly superior to the Python implementation in time performance. In the final DEVSTONE LARGE example, pypdevs threw exceptions and was unable to finish the simulation.

9 Performance Comparison: ADEVS and cDEVS

9.1 Traffic and Queue

We only succeeded in implementing the Queue and Trafficlight models in ADEVS. From running simulations, we can see that ADEVS is superior (factor 50) to cDEVS when running the Queue example and shows equal performance in the small traffic light models. There are several reasons why ADEVS can and should be faster than cDEVS. For example, we do not use optimal allocation of models on the nodes. ADEVS has also been optimized more than cDEVS.

10 Software & Libraries

10.1 Compilation and Development

The following software and tools were used during the development of the project

Environment	The Eclipse IDE was used as programming environment.
Compilation	Compilation is generally performed on OS X and Unix with the tools presented earlier in this report.
Testing	Testing was implemented using the Google library gtest. More information on this can be found in the serialization prototype report, and the user manual.
VC	Bitbucket and Git were used a Version control.
CI	Jenkins was used as the Continuous Integration server.